Comp. & Maths. with Appls. Vol. 8, No. 4, pp. 319–327. 1982
Printed in Great Britain.

0097–4943/82/040319–09$03.00/0
Pergamon Press Ltd.

DETECTION OF CONNECTIVITY FOR REGIONS REPRESENTED BY LINEAR QUADTREES†

IRENE GARGANTINI
Department of Computer Science, The University of Western Ontario, London, Ontario, Canada N6A 5B9
Communicated by Yuddell L. Luke

(*Received December* 1981)

Abstract—An algorithm is given for the determination of all connected components of a given region. The data structure used to represent the region (or image) is the linear quadtree, a recently introduced structure which basically consists of a sorted array of quaternary integers. A linear quadtree is built in a way similar to that of a regular quadtree but contains, as elements, only black nodes. The proposed algorithm has computational complexity proportional to $12nN$, where n is the resolution parameter of the image, and N is the number of black nodes. Space complexity is proportional to $6N$. The algorithm is shown to compare favourably, both in terms of space and time, with the existing method due to H. Samet.

1. INTRODUCTION

The determination of connected regions is a problem arising in various fields of computer science, like image processing, cartography, numerical analysis and computer graphics. A region (or image) is given, in general, as a set of black unit-square pixels arranged on a $2^n \times 2^n$-raster where n is the resolution parameter or degree of refinement of the screen. To facilitate operations on regions (like contour determination, union, intersection, rotation, etc.), a compact hierarchical structure is often used, called quadtree [1, 2] which represents both the region (black pixels) and its background (white pixels) with respect to the $2^n \times 2^n$-array. To identify a quadtree node, six fields are required, four of which are pointers to the sons, one is a pointer to the father and the other carries information on the node colour.

In [3, 4] the author has introduced a new data structure, called "linear quadtree", which condenses all information normally contained in the above-mentioned six fields into only one. The key of the new data structure consists of replacing a black quadtree node with a quaternary integer, each digits of which represents the successive quadrant subdivision ($n = 1, 2, \ldots$) with the following encoding: the North-West quadrant is encoded with 0, the North-East quadrant with 1, the South-West quadrant with 2 and the South-East quadrant with 3. For instance, if a square pixel belongs to the South-East quadrant in the first subdivision, to the South-West quadrant in the second and to the North-West quadrant in the third and last subdivision, then 320 represents the pixel. Condensation (aimed at grouping four pixels belonging to the same quadrant(s)), takes place by means of a special marker X. For example, if all four codes 320, 321, 322, 323 are present, they are grouped together to form the "mixed-quaternary" code $32X$; if $30X$, $31X$, $32X$, and $33X$ are all present, they are replaced by $3XX$, and so forth. The region shown in Fig. 1 is associated with the quadtree representation given in Fig. 2, while the corresponding linear quadtree is simply

$$003\ 021\ 023\ 03X\ 122\ 21X\ 3XX.$$

As the reader can see, the digits (left to right) of the quaternary code supplies the path from the root to the given node, while the order in which the quaternary integers are sorted corresponds to the postorder traversal of black nodes in the quadtree.

With the introduction of linear quadtrees, an image, formed of black pixels, can be represented by a dynamically built array of sorted mixed-quaternary codes. To achieve this X must be encoded with an integer > 3 as explained in [3].

†This work was supported by the Canadian Government under the NSERC Grant No. A7136.

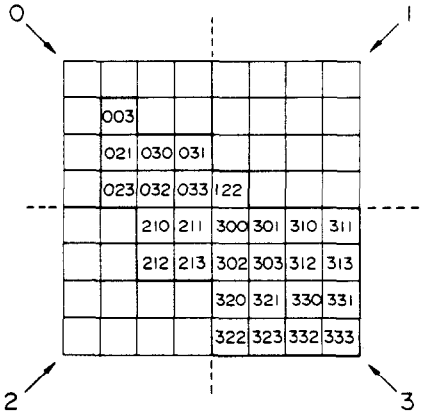

Fig. 1. Quadrant labelling: generation of quaternary codes for a given region.

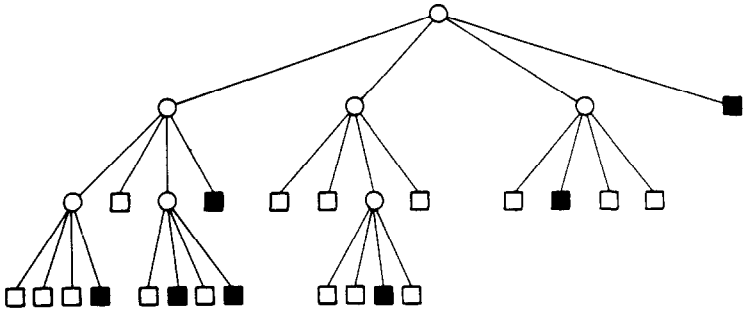

Fig. 2. Quadtree for region of Fig. 1.

The advantages of linear quadtrees with respect to quadtrees can be summarized as follows: (i) space complexity is remarkably decreased [3]; (ii) worst-case time complexity of the related algorithms (like contour determination, rotation, background evaluation etc.) is, in most cases [4], improved; (iii) the algorithms mentioned in (ii) are easy to design and optimize [4].

The extension of linear quadtree to three-dimensional images is presented in [5].

Scope of this paper is to present an algorithm for labelling all connected components of a two-dimensional region. The input is represented by a sorted array formed of mixed quaternary codes. The output is formed of $K \geqq 1$ connected regions, given as sets of quaternary codes.

Space complexity of the proposed algorithm is bounded by $6N$, where N is the number of black nodes forming the given image. Worst-case time complexity is proportional to $12nN$.

In the sequel we shall adopt the following terminology and notation. n, as explained before, is the degree of refinement of the image; M is the number of nodes in a quadtree; N is the number of black nodes forming the given region; K is the number of connected regions. LINQUAD is the array of quaternary codes representing the input image. K BUCKETs of quaternary integers represent the K disjoint output regions.

2. SAMET'S METHOD

In [6] Samet presents a method for labelling all connected components of a region represented by a quadtree. The method consists of three phases. The first one labels all black nodes while recording adjacencies between any pair of them, the second one processes equivalences, and the third one labels all connected regions with the same identifier. The computational complexity of the algorithm—based on an average estimate for determining adjacencies, and on worst-case bounds for all other processes—is given by $(14nN + N + 2)$. Combining the first and second phases may result, in practice, in a faster algorithm.

Space complexity depends on M, the total number of nodes (black, white, and grey) of the input quadtree and on the number of nodes of the dynamic structure (forest or else) used to process equivalence pairs. Since the latter is proportional only to N and N is expected to be

much smaller than M, the dominant term of Samet's algorithm depends on M. If we assume that one computer word can hold two pointers or one pointer and two identifiers (for colour and label), then space complexity, measured in terms of number of memory locations, is proportional to $3M$.

In the next section we present a different approach which produces a better "all-around worst-case" time complexity and requires a number of memory locations which is proportional only to N.

3. AN ALTERNATIVE TO SAMET'S METHOD: INFORMAL DESCRIPTION

In this approach we visit each node of the input image and test its four adjacent nodes: when an adjacency relation between any two elements is found, these are put into the same bucket. If no adjacency exists, a new bucket is created. After the last node is visited, $K \geqq 1$ buckets emerge, each containing all (and only) nodes belonging to a single connected region.

Let us try to explain the algorithm (formally given in the Appendix), by means of an example. Consider the region given in Fig. 3 and the corresponding sorted array (named LINQUAD) formed of the mixed quaternary codes given below:

0231, 0303, 0312, 0313, 0320, 0322, 0331, 1032, 1210, 1211, 1212, 1220, 1221, 1302, 1321, 1331, 1332, 1333, 2100, 2101, 311X, 313X, 33XX.

Our goal is to supply $K = 6$ connected regions as sets of quaternary codes. The algorithm we propose uses two auxiliary data structures, a linked list (called BUCKET) and a queue (called QUEUE). For each region BUCKET contains all nodes belonging to a connected region and QUEUE all quadruples representing the nodes adjacent (in the four principal directions) to the nodes of BUCKET. CURSOR is an array pointer which scans LINQUAD, skipping the elements which have been already found to belong to a given region. For each new value of CURSOR, a BUCKET and a QUEUE are created. In the first element of BUCKET we store the node whose subscript is CURSOR, while the corresponding four adjacent nodes are stored in QUEUE. The first node of QUEUE—identified by HEAD—is tested. If it belongs to LINQUAD, that node is added to BUCKET, the value of the node in LINQUAD is replaced by the label "BELONG" and its four adjacent nodes are inserted into QUEUE (from the end, of course). HEAD is advanced to the next element and the old pointer is freed. HEAD contains

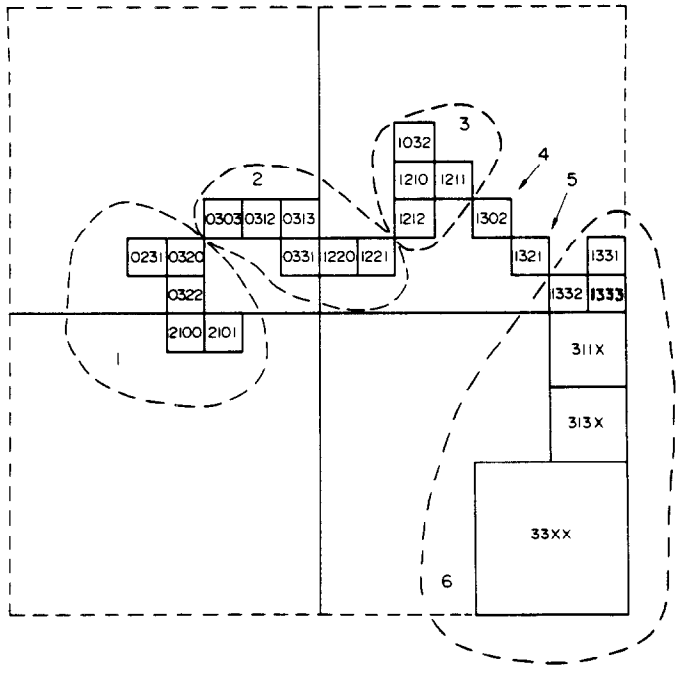

Fig. 3. Detection of connected components.

now the first adjacent node; the latter is tested to see whether or not it belongs to LINQUAD. If it does, it is stored into BUCKET and its four adjacent nodes are inserted into QUEUE; otherwise HEAD is advanced to the next element. This "inner" process is repeated until QUEUE is empty. At this point BUCKET contains all the nodes belonging to a connected region; they can be output so that BUCKET and QUEUE can be returned to the free space. The "outer" process continues by incrementing CURSOR until all elements of LINQUAD are visited. At this point all connected regions have been identified.

In the example given above, for instance, for CURSOR = 1 we recognize the connected region formed of nodes

0231, 0320, 0322, 2100, 2101;

for CURSOR = 2 another region is given by

0303, 0312, 0313, 0331, 1220, 1221,

and for CURSOR = 8 we have the region

1032, 1210, 1211, 1212.

For CURSOR = 14, 15 we have two distinct regions, each represented by only one square pixel (1302 and 1321 respectively). Finally for CURSOR = 16 we get the last region (as shown in Fig. 3)

1331, 1333, 311X, 1332, 313X, 33XX.

The algorithm (named CONNECT-REG) is given in the Appendix. It assumes that five additional procedures are available: the first, called BINSEARCH, performs a binary search on LINQUAD between (CURSOR + 1) and N. For an elegant implementation of a binary search, see, for instance, [7]. The other four procedures, denoted E-ADF-TO (element), S-ADJ-TO (element), W-ADJ-TO (element) and N-ADJ-TO (element), are implementations of the algorithms given in [3].

4. ANALYSIS OF THE PROPOSED ALGORITHM

Let $N_1, N_2, \ldots, N_K$ be the number of black nodes of the 1st, 2nd, ..., Kth component (respectively). Let $\bar{N} = \max\{N_1, N_2, \ldots, N_K\}$ and $S(N)$ be the space complexity of the structures used by the algorithms, measured in terms of memory locations. Here we assume that a value and a pointer can be packed into a single computer word. We have

THEOREM 1

The space complexity of CONNECT-REG is given by $S(N) = N + 5\bar{N} + \text{constant} \leqq 6N + \text{constant}$.

Proof. The first term is supplied by the length of the input array (N); the other term is supplied by the dimensions of BUCKET (consisting of $\bar{N}$ memory locations) and QUEUE (consisting of $4\bar{N}$ memory locations). The bound is sharp for $K = 1$.

Time complexity $T(N)$ is evaluated by remembering that an "easy" (if not a good) bound for $\log_2 N$ is $2n$ since $N < 2^{2n}$.
We have

THEOREM 2

The worst-case time complexity of CONNECT-REG is given by

$$T(N) = 4N \log_2 (N-1) + 4nN + \text{constant} < 12nN + \text{constant}.$$

Proof. The first term originates from the comparison of all adjacent nodes in QUEUE with

the nodes of the input array LINQUAD: this is a binary search carried out on an array of length at most $(N-1)$; the second term is due to the generation of the four adjacent nodes, according to the algorithms give in [3]. As shown there, the production of an adjacent node can be carried out, in the worst case, in time proportional to n.

Finally, we note that $T(N)$ would be smaller in the presence of a parallel model of computation, since then all four adjacent nodes could be evaluated simultaneously.

5. COMPACT REPRESENTATION OF MIXED-QUATERNARY CODES

To speed up comparison between two nodes, one of which is covered by the other, the representation given in the introduction can be modified as follows. Represent each node by a quaternary integer (i.e. with digits 0, 1, 2, 3 to the base 4) preceded by a counter expressing the number of X's. Thus $X = 320XX$ and $X' = 32010$ would be represented (respectively) as

2	320		0	32010

To determine whether or not X covers X' (or vice versa) we first compare the two counters. If the X-counter is greater or equal than the counter of X' we shift X' (X-counter-X'-counter) quaternary positions to the right and subtract X' from X. If the difference is 0, then X covers X', otherwise it does not. In the other case (i.e. X-counter $<$ X'-counter) we interchange the roles of X and X'. Thus testing two nodes requires only two comparisons, one shift and two differences. Both the counter and the quaternary digits can be packed in one computer word for reasonable values of n.

6. COMPARISON WITH THE ALGORITHM FORMULATED FOR QUADTREES

As shown in Section 4, the algorithm presented in this paper is faster than the one based on quadtrees; another interesting feature is that each connected component identified by CONNECT-REG could be given, with only minor modifications, as a linear quadtree. What remains to be done is a comparison of the performance of the two algorithms in terms of space requirements. To do this we need upper and lower bounds for the number of nodes M expressed as functions of n and N. Let us assume $N \geqq 3$. The first is given by Samet as per

LEMMA 1

The upper bound on the number of nodes M of the quadtree is $4nN + 1$ (see, for instance [6]).

The second will be derived here below.

LEMMA 2

The lower bound on the number of nodes M is $(N + n + 1)$.

Proof. We apply the technique of local optimization and explore possible changes which can lead to the (global) optimal case. First consider the case $N = 3n$. For $n = 1$ the structure giving the minimum number of nodes consists of the root, and four terminal nodes (one white, three black). For $n > 1$ the structure giving the minimum total number of nodes at a given level (different from last) consists of a grey node and three black nodes. The last level is formed of a white node and three black nodes. To optimize the entire structure, we now try to replace a given node with one of different colour. However, if we replace a black node with a grey node we increase the total number of nodes and so we do if we change a black node into a white one. Also in the case of a grey node, any change would produce a larger value of M. Therefore, $(N + n + 1)$ is the minimum number of nodes for $N = 3n$.

The other two cases ($N = 3n - 1$ and $N = 3n - 2$) are treated in the same way: the colour of the last level nodes changes, but the total number of nodes remains the same.

With the estimates given in Lemmas 1 and 2 we can now measure the gain (expressed in terms of memory locations and with respect to quadtrees) when: (i) we store black pixels using linear quadtrees and (ii) we apply algorithm CONNECT-REG for the detection of connectivity.

Let us first condense our findings under the form of theorems, as follows:

THEOREM 3

Complexity required by a linear quadtree is N, while for a regular quadtree the number of memory locations, denoted $S(\text{quad})$, varies according to

$$3(N+n+1) \leqq S(\text{quad}) \leqq 3(4nN+1) \tag{1}$$

Proof. Left and right sides of (1) follow from the previous Lemmas and the assumption, made at the beginning, that three memory locations are required for each quadtree node.

THEOREM 4

Space complexity required by the data structures used in CONNECT-REG is, at most, $6N$, while for a regular quadtree the number of memory locations, denoted here $S(\text{labelling})$, varies according to

$$3(N+n+1)+2N \leqq S(\text{labelling}) \leqq 3(4nN+1)+2N. \tag{2}$$

Proof. The first statement follows from Section 4, while (2) follows from Lemmas 1 and 2 and the estimate of $2N$ for the dynamic structure supporting the equivalence merging algorithm (see, for instance, [8]). This latter is assumed implemented using an array of N pointers for the N nodes of the forest, for a total of $2N$ memory locations.

Finally, Table 1 illustrates the reduction (if less than 100) or the increase (if more than 100) implied by the use of linear quadtrees with respect to regular quadtrees for $n = 6, 8, 10$. Columns (a) and (b) refer to storage alone, while columns (c) and (d) refere to space requirements in connection with the algorithm proposed here for the detection of connectivity.

Table 1. *Memory locations required by linear quadtrees with respect to quadtrees*

n	(a) min	(b) max	(c) min	(d) max
6	25%	1.4%	100%	8.3%
8	25%	1%	100%	6.2%
10	25%	0.8%	100%	5%

The least gain occurs when the number of total nodes in the quadtree is the least possible, i.e. when $N = 3n$ or $3n-1$ or $3n-2$, so that $N = 4n+1$. Therefore, the ratio (number of memory locations for linear quadtree/number of memory locations for quadtree) is close to $3n/3(4n+1) \cong 0.25$ (column (a) of Table 1). The most gain occurs when $M = 4nN+1$ and the above ratio is close to $1/(12n)$. Some values are given in column (b) for $n = 6, 8, 10$. Similar evaluations are given in columns (c) and (d) for the algorithm CONNECT-REG using bounds (2).

As the reader can see, for a $1{,}024 \times 1{,}024$ screen, the use of linear quadtrees can save between 75%–99% of the memory locations required by regular quadtrees. For the detection of connectivity, CONNECT-REG can use the same number of memory locations as Samet's method, or can save, in the most favourable cases, up to 95% of the memory locations used by. quadtrees.

REFERENCES

1. G. M. Hunter and K. Steiglitz, Operations on images using quadtrees. *IEEE Trans. Pattern Anal. Mach. Intell.* **1**, 145–153 (1979).
2. C. R. Dyer, A. Rosenfeld and H. Samet, Region representation: boundary codes from quadtrees. *Comm. ACM* **23**, 299–314 (1980).
3. I. Gargantini, An effective way of storing quadtrees. *Comm. ACM.* Conditionally accepted.
4. I. Gargantini and Z. Tabakman, Linear quad- and oct-trees: their use in generating simple algorithms for image processing. *Proc. Graphics Interface* '82. To appear.
5. I. Gargantini, Linear oct-trees for fast processing of three-dimensional objects. *Comput. Graph. Image Proc.* To appear.

6. H. Samet, Connected component labelling using quadtrees. *Comput. Sci. TR-756*. University of Maryland, College Park (1979).
7. M. J. Augenstein and A. M. Tanenbaum. *Data Structures and PL/I Programming*, pp. 220–223. Prentice-Hall, Englewood Cliffs (1979).
8. D. E. Knuth, *The Art of Computer Programming*, Vol. 1, pp. 353–355, 360. Addison-Wesley, Reading, Mass. (1973).

APPENDIX

Here below we give the specifications of the algorithm CONNECT-REG (LINQUAD,N) in view of its implementation on a computer.

input: the sorted array of quaternary codes, LINQUAD, and its length.

output: sets of quaternary codes, identified by BUCKET. Each set represents a connected component.

data structures used: an array (LINQUAD), a (singly) linked list (BUCKET) and a queue (QUEUE). CURSOR is the subscript for LINQUAD, scanning the array from the first to the last element. BUCKET is the head of the linked list; each node consists of two fields, one for the quaternary code (VALUE) and the other for the pointer (NEXT). QUEUE is identified by two subscripts, called HEAD and TAIL. The implementation of QUEUE (array or linked list) is left to the programmer.

auxiliary procedures required: a routine to perform a binary search (BINSEARCH), and four functions for finding adjacent nodes [3], denoted E-ADJ-TO, S-ADJ-TO, W-ADJ-TO, N-ADJ-TO.

suggested programming languages; any high-level language with dynamic allocation and freeing of memory (PL/I, PASCAL, ADA,...).

```
  procedure CONNECT-REG(LINQUAD,N)
(*this procedure outputs all connected components of the
  input region LINQUAD*)
    begin
        CURSOR=1
        while CURSOR less than N begin
              while LINQUAD (CURSOR) not equal to 'BELONG' begin
(*create linked list BUCKET*)
                        CREATE BUCKET,FIRST
```

```
                        VALUE OF BUCKET = LINQUAD(CURSOR)
                        NEXT OF BUCKET = NIL
                        FIRST = BUCKET
(*create data structure QUEUE*)
                    CREATE QUEUE    IDENTIFIED BY HEAD AND TAIL
                        QUEUE (HEAD) = E-ADJ-TO (LINQUAD(CURSOR))
                        QUEUE (HEAD+1) = S-ADJ-TO (LINQUAD(CURSOR))
                        QUEUE (HEAD+2) = W-ADJ-TO (LINQUAD(CURSOR))
                        QUEUE (HEAD+3) = N-ADJ-TO (LINQUAD(CURSOR))
                        TAIL = TAIL+3
                    while QUEUE not EMPTY begin
                        COMPARAND = QUEUE(HEAD)
(*apply binary search to LINQUAD between (CURSOR + 1) and N skipping
elements marked "BELONG'.  If search succeeds, it supplies 'an'
element LINQUAD (SEC-CURSOR) equal to, or belonging to, or covering
COMPARAND, together with the subscript SEC-CURSOR;  if search fails
it sets BINSEAR equal to NIL*)
                        call BINSEARCH (LINQUAD,COMPARAND,CURSOR,
                                        BINSEAR,SEC-CURSOR,N)
                        if (BINSEAR not NIL) then begin
 (*update BUCKET and QUEUE *)
                        call ALPHA (LINQUAD (SEC-CURSOR),BUCKET,
                                        QUEUE, CURSOR, FIRST,TAIL)
                        LINQUAD (SEC-CURSOR) = 'BELONG'
 (*free memory location in QUEUE identified by HEAD*)
                        HEAD = HEAD+1
                        end
              end of while
         output BUCKET
 (*free linked list and queue *)
         FREE BUCKET, QUEUE
         end of while
         CURSOR = CURSOR+1
     end of while
 end

                    ---------------
 procedure ALPHA(THIS,BUCKET,QUEUE,CURSOR,FIRST,TAIL)
 (*insert THIS into BUCKET and its adjacent nodes into QUEUE.
```

```
 BUCKET does not need to be passed explicitly *)
    begin
        CREATE NODE
               VALUE OF NODE = THIS
               NEXT OF NODE = NIL
               NEXT OF FIRST = NODE
               FIRST = NODE
(*create four new memory locations at the end of QUEUE*)
               QUEUE (TAIL+1) = E-ADJ-TO (THIS)
               QUEUE (TAIL+2) = S-ADJ-TO (THIS)
               QUEUE (TAIL+3) = W-ADJ-TO (THIS)
               QUEUE (TAIL+4) = N-ADJ-TO (THIS)
               TAIL = TAIL+4
    end
```